Porting to the Intel Xeon Phi: Opportunities and Challenges

C. Rosales
Texas Advanced Computing Center
The University of Texas at Austin
J.J. Pickle Research Campus, Building 196
Austin, Texas 78758-4497
Email: carlos@tacc.utexas.edu

***Abstract*—This work describes the challenges presented by porting code to the Intel Xeon Phi coprocessor, as well as opportunities for optimization and tuning. We use micro-benchmarks, code segments, assembly listings and application level results to illustrate the key issues in porting to the Xeon Phi coprocessor, always keeping in mind both portability and performance.**

While executing code on the Xeon Phi in native mode is fairly straightforward it can be a challenge to achieve good performance. The complexity of optimization increases as one introduces offload, distributed offload, or symmetric execution modes. We will initially focus on the fundamental issues that can prevent acceptable performance in native execution, and then address the key issues in data transfers due to either offloaded regions or MPI exchanges with the host CPU. Some of the issues are of a generic nature and affect any code using heterogeneous execution – PCIe bandwidth bottleneck –, and others are specific to the Xeon Phi and its software environment – Host/MIC MPI exchanges. We will also make an effort to indicate which issues are specific to this platform and which are of general applicability. In particular we will draw comparisons between the data management models in the Intel Xeon Phi and in the NVIDIA CUDA environment.

I. Introduction

With the drive to reach higher computational density the largest and most powerful computing systems in the World have been adopting alternatives to the traditional CPU as one of their main components for some time. NVIDIA GPUs have powered many of the top ranked TOP500 systems since 2008, and in the latest published list [1] two systems with Intel Xeon Phi coprocessors have claimed positions 1 and 6 in the list.

While it is clear that the need to increase efficiency will continuously drive changes in hardware, it is important to understand these new systems, their advantages and limitations, and the effort that it takes for a researcher to port his code efficiently to the new platforms.

The Intel Xeon Phi platform presents an opportunity for researchers using simulation techniques because unlike other coprocessors and accelerators it does not require learning a new programming language or new parallelization techniques. The programming paradigm for this platform is based on C/Fortran and OMP/MPI standards, and will be familiar to developers already working in multicore systems [2].

This does not mean that achieving good performance on this platform is simple. The hardware, while presenting many similarities with other multicore systems on the surface, has its own characteristics and idiosyncrasies that must be understood in order to port code in an efficient manner.

Recent studies show that the Xeon Phi coprocessor is capable of executing a variety of kernels with high performance [3], [4], [5], and even some applications [6], [7]. Our objective in this work is to show the process followed to reach that performance when dealing with an application, and highlight the elements that affect performance in a critical manner.

II. Many Integrated Core Architecture

All the tests reported in this work where performed on the Intel Xeon E5-2680 2.7GHz host CPUs and the Intel Xeon Phi SE10P coprocessors in the Stampede nodes at TACC [8].

The Intel Xeon Phi SE10P (silicon rev B0), which we refer to as MIC or Xeon Phi interchangeably in this paper, is a 61 core, x86 based, SMP on a chip coprocessor. It has 64-bit support, and 512-bit SIMD capability. Each core has a 32 KB L1 data cache, and a 512 KB L2 cache that is remotely accessible to all other cores via a high-speed bi-directional ring. The cores support four hardware threads for a total of 244 threads on a single MIC, and run at a clock speed of 1.09 GHz. The presence of multiple threads is necessary in order to hide the latency characteristics of in-order execution architectures, and for most scientific applications the best performance is achieved when using at least two hardware threads per core. The MIC card communicates via PCIe with the host CPU, and has its own memory - 8GB GDDR5 at 5.5 GT/s - and runs its own lightweight Linux operating system called BusyBox.

From the point of view of the programmer the MIC is a processor with a 512-bit wide vectors and a fully coherent cache. While designed with high thread counts in mind, the MIC architecture supports also MPI and hybrid execution modes.

Since the software stack is currently evolving, we feel it is important to be precise regarding the particular versions of the compiler and system drivers used when collecting our results. We used the Intel Manycore Platform Software Stack (MPSS) version 2.1.6720-13, and the Intel Compiler version 13.1.1.163.

III. Porting an LBM code

The code chosen for this work was a multiphase Lattice Boltzmann Code based on the Free Energy method of Zheng,

Shu and Chew [9]. This code simulates the flow of two immiscible, isothermal, incompressible fluids with great spatial and temporal detail. Basically there are two arrays, f and g, which represent the phase and the momentum of the fluids and obey a linearized version of the Boltzmann Equation, with collision terms that follow the BGK approximation. For details on the model we refer the interested reader to [10] and [11]. The only deviation from the traditional solution of the LBM equations that is present in the Zheng model is an extra smoothing term in the streaming of the f components, which allows for higher density ratios to be used between the two fluids. This prevents the use of a merged collision-stream step for f. A merged collision-step is still used for g because it saves half the memory traffic due to the calculation and propagation of g components.

All reported timings were taken for 100 iterations of a 240x240x240 test case using three individual runs and taking the average of the three runs. Very little variability between runs was observed, so this number of runs was considered sufficient. We will often describe performance in terms of MLUPS, or millions of lattice updates per second, a typical measure of performance for LBM codes that is simply calculated as:

$$MLUPS = \frac{(Mesh\ Volume) \times (Time\ Steps)}{Execution\ Time} \times 10^6$$

The run size of 240x240x240x was chosen because it corresponds to a typical size per node in a production run, and because larger runs would require additional coding work since they would not fit in the coprocessor memory.

All versions of the code code were run in both the host CPUs and the Xeon Phi coprocessor in Stampede. Unless otherwise stated each run employed 16 threads in the host CPUs, using one thread per core, and bound each thread to a core using the Intel environmental variable `KMP_AFFINITY=compact,granularity=fine`. Similarly, for the native Phi coprocessor runs we used 240 threads and bound the threads using `KMP_AFFINITY=balanced,granularity=fine`. As mentioned before, it is important to remember that a Xeon Phi thread can only issue a vector instruction every other cycle, so having at least two threads per core is required to make full use of the vector unit.

A. The original code

The original code had a very simple structure. The evolution equation for the fluid was solved in the traditional LBM manner [12], using a set of local operations (the collision step) followed by a set of memory moves (the streaming step) plus a few boundary corrections.

After initialization a main loop was executed where four functions where run in order:

- Collision: Collision terms are calculated for f and g. Values of g are streamed
- PostCollision: Boundary values are corrected for f
- Stream: Values of f are relaxed and streamed

Hbt

TABLE I. EXECUTION WITH PARALLELIZED `COLLISION` ONLY

Function	Time (s)	% Run Time
Collision	44.40	24.82
PostCollision	1.07	0.60
Stream	123.18	68.85
PostStream	10.29	5.75

- PostStream: Streamed values of f and g are corrected at the boundaries

The data structure in the original implementation was very simplistic, with one array for each distribution function, f(i,j,k,v,buf) and g(i,j,k,v,buf). Here (i,j,k) are indices representing the x,y and z coordinates; v represents each of the discretized velocity components; and buf is 0 or 1 depending on which side of the double-buffer array the data is being read or written.

Since the `Collision` function contained all the floating point work we added OpenMP statements to each of the three outer loops within: the update of the phase value in the array "phi"; the calculation of the gradients and Laplacian of "phi"; and the collision and stream of each of the 19 g components. This meant that we were distributing the z values among the OpenMP threads. We thought this would provide a sensible performance since in a traditional CPU the collision function takes approximately 95% of the execution time for a sequential execution.

The initial performance of this code in the Phi coprocessor was very low, with an output of only 7.7 MLUPS. The same code running on 16 threads on the host CPUs performed at a much higher 32.9 MLUPS, more than 4 times faster than on the coprocessor. While disappointing, this represented the outcome of the most naive attempt possible at porting the code to the MIC platform. We proceeded to time each of the four functions in the main loop, obtaining the results in Table I.

What these results indicate is that a single thread on the coprocessor is not able to achieve sufficient bandwidth to memory and so the sequential section of the code in the `Stream` function became the bottleneck for execution. The next step was clear: to eliminate the memory bandwidth bottleneck in the `Stream` function.

B. Improving bandwidth bottlenecks

While out initial focus had been on the`Collision` function, the timings from our experiments clearly indicated that a single thread on the coprocessor was unable to achieve the memory bandwidth necessary to perform the memory copies in the `Stream` function efficiently, creating a new bottleneck in the code.

The next change was to introduce OpenMP statements in the `PostCollision`, `Stream`, and `PostStream` functions, all of which perform a large number of memory moves but no significant floating point work. Once more we added OpenMP pragmas to the outermost loops in each of these functions. In the case of the `PostCollision` and `PostStream` functions, which contain many loop sections, we defined a single OMP parallel section and then used `!DIR$ OMP DO` statements before each parallelized loop in order to avoid the unnecessary overhead.

TABLE II. EXECUTION WITH PARALLELIZED STREAM

Function	Time (s)	% Run Time
Collision	44.07	96.26
PostCollision	0.03	0.06
Stream	1.39	3.04
PostStream	0.29	0.63

TABLE III. EXECUTION USING SOA

Function	Time (s)	% Run Time
Collision	8.89	82.31
PostCollision	0.03	0.28
Stream	1.59	14.72
PostStream	0.29	2.68

The performance of the new code on the Phi coprocessor increased to 29.7 MLUPS, nearly four times the speed of the initial version of the code. Detailed timings, shown in Table II show that the hotspot in the code moved back to the collision function. Using 240 threads the stream function sped up by a factor 89x, which gives an idea of how important it is to thread code sections that perform significant amounts of data movement in memory.

This improvement also translated, to a smaller degree, to the host CPU. The new code achieved 52.4 MLUPS, an improvement of 1.6x over the previous code. The improvement was relatively smaller per thread that in the Phi coprocessor because the threads on the host cores are more powerful, and less of them are needed to saturate the memory bandwidth to the socket.

C. Improving vectorization

While fixing the single thread bottleneck improved the code performance on the Phi coprocessor, its overall performance was still low when compared to running on the host CPU. We turned then to analyzing the code vectorization. The vector report for the`Collision` function indicated that the innermost loops in the first two sections of the function were vectorized but the third – and most computationally intensive – loop was not vectorized due to a flow-type data dependence.

We compiled the code twice, once using `-O3 -openmp -mmic` and another using `-O3 -openmp -no-vec -mmic`. The second set of flags prevented the compiler from vectorizing the code, allowing for a direct comparison of the vectorized and non-vectorized versions of the code. The result was that the vectorized code achieved 29.7 MLUPS, as mentioned in the previous section, and the non-vectorized code 27.9 MLUPS. Since a fully vectorized code could potentially achieve a speedup factor of 8x over a non-vectorized double precision code, this indicates a very inefficient vectorization. This, in turn, could be due to memory access patterns in the code, or simply the lack of vectorization in the most computationally intensive section of the code as indicated by the compiler report. Including `IVDEP` pragmas in the collision function did not achieve vectorization of the third section of the function.

Since there was no true data dependence we suspected that the compiler was being confused by the way the collision function accessed data in the double-buffered arrays for f and g. We decided to simplify the data layout and change from a single array containing all velocity components to multiple arrays, one for each velocity component, and each array with a single dimension index. This is equivalent to moving from a setup akin to an array of structures (AOS) to a setup that is more like an structure of arrays configuration (SOA).

The output from the vector report on the compilation of the new code confirmed that in this case all three sections of the collision function were vectorized, and the performance of our test case increased to 127.7 MLUPS, a factor over four times speedup from the previous version. We tested again a non vectorized version of this code and the resulting performance was 29.7 MLUPS. This indicates that the code is using the vectorization capabilities of the Xeon Phi at approximately 54% efficiency, which means there is still room for improvement in the code but also that the changes introduced were very effective. The relatively low vectorization is most likely due to memory access issues that prevent the coprocessor from executing a vector operation per cycle. The author suspects these issues to originate in the large number of simultaneous memory streams in the LBM code, which are upwards of fifty in some sections of the `Collision` function, and is currently working on alternative code structures that can alleviate this issue.

When the same code is executed on the host CPU a performance of 50.1 MLUPS is achieved, which is not significantly different from the previous value. The key consequence of these code changes is that by improving the use of the vector units in the Phi coprocessor we now have a performance that far surpasses that of the host CPU, with a single coprocessor providing performance equivalent to that of 2.5 dual socket Xeon E5-2680 nodes.

It is important to notice that this change from AOS to SOA is of benefit not only for the Xeon Phi coprocessor, but also for other architectures. For example, when writing CUDA codes it is critical to achieve coalesced memory accesses in order to obtain as high an effective memory bandwidth as possible, and a code with an SOA structure can achieve coalesced access much easier than a code with an AOS structure.

D. Scaling and affinity

Because of the large number of threads supported by the Phi coprocessor, affinity settings can change performance in a critical manner. Using the Intel KMP affinity control settings one can select a `compact`, `scatter`, or `balanced` thread binding. The `compact` setting binds threads to consecutive processors, while `scatter` uses a round-robing placements, and `balanced` attempts to spread threads as much as possible, but keeps consecutive threads close in terms of hardware placement. Figure 1 shows an example where 8 threads are placed using each of these settings on a figurative 4 core system. Notice that the compact setting, while exploiting locality as much as possible, leaves the system underutilized.

Figure 2 shows the performance of the native code using the three basic affinity settings. Notice how `balanced` and `scatter` provide the same performance as long as we use a single thread per core, as expected, but the `balanced` setting is significantly better when all cores are in use and there are multiple threads per core. Because this code produces a high amount of memory traffic, leaving cores unused is

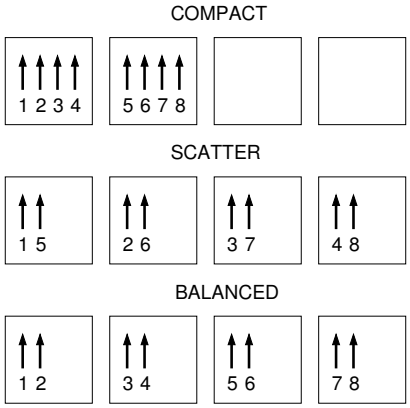

Fig. 1. Effect of different thread affinity settings for a figurative 4 core system supporting 4 hardware threads per core.

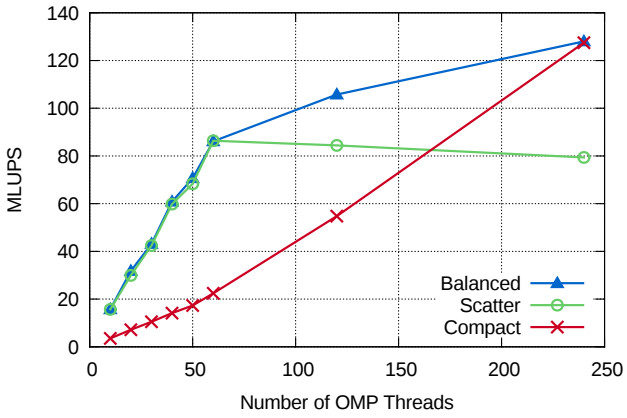

Fig. 2. Effect of different affinity settings on the LBM code performance. The balanced setting provides the best results overall.

not beneficial, as shown in the lower performance of the `compact` setting. Both `compact` and `balanced` settings produce the same thread distribution when the coprocessor is fully populated, and so they provide the same performance for the case that runs 240 threads. The differences in performance for different affinity settings highlight the importance of thread binding in this newer multicore systems.

E. Exploring alignment issues

Since the Phi coprocessor does not support non-aligned vector operations, a significant amount of preparation work must be done by the system when non-aligned accesses occur in the code, reducing the overall efficiency. While the LBM code suffers from non-avoidable misaligned accesses in many sections there are some parts where all accesses are aligned, even accounting for the OpenMP loop breakup. This is an important point. To ensure correct code is generated when one uses compiler hints such as `VECTOR ALIGNED` or `SIMD` it is not only necessary that an array is aligned, but that the access from every thread is aligned as well.

The only section of the code that verifies these conditions is the first section of the collision function, where the order parameter, ϕ, is calculated from the values of the f distribution function. Besides including the `VECTOR ALIGNED` directive

TABLE IV. EFFECT OF ALIGNMENT ON `COLLISION` LOOPS

Function	Non-aligned	Aligned
Loop 1	0.70	0.68
Loop 2	0.81	0.83
Loop 3	7.38	7.46
Total	8.89	8.97

in this loop we also allocated space for our arrays ranging from -7 to the maximum size instead of starting at 0. This was done because 0 denotes the ghost cells used to fix the values in the boundary in the `PostCollision` and `PostStream` functions, and we want the access to element 1 in the array (the first active node) to be aligned to a 64 byte boundary because the main computation loops in `Collision` go from 1 to the maximum size. We also added the `-align array64bytes` to the compilation flags.

The results were disappointing and, at the beginning, confusing. The overall performance dropped to 95.1 MLUPS. We looked at the assembly code generated for the order parameter calculation loop and the results indicated that the compiler had indeed made good use of the knowledge that the access was aligned, avoiding all instances of `vloadunpackhd` and `vloadunpackld` that are commonly observed in non-aligned calculations. The code was much tighter, too, with 277 lines of assembly generated instead of 656 for the non-aligned case.

To investigate why this change had a negative effect in the overall performance we instrumented the code with call to `clock_gettime()` around the three sections in the `Collision` function. The results, collected in Table IV, showed that while the order parameter calculation loop was faster in the aligned version of the code, the second two sections of `Collision` became slower. This was most likely due to a change in prefetching behavior or the actual data layout in memory for the aligned case. It is possible that an alternative structure for the `Collision` function would change this results, but we have not had success in our attempts so far. All attempts to break up the larger collision loop into smaller code sections have turned out to be unfavorable in terms or data reuse and number of memory accesses.

Another interesting detail regarding the use of the `ALIGNED` directive is that if two consecutive loops have the `VECTOR ALIGNED` and `VECTOR` directives the Intel compiler can sometimes merge both loops during the optimization stage and then proceed to assume that they are both aligned, leading to a segmentation fault during execution. While irrelevant to performance, this is a detail to keep in mind when using such directives. The behavior can be prevented by the use of a `NOFUSE` directive in the second loop.

F. Simultaneous host and coprocessor execution

To write a code that was easily run on both the coprocessor and the host without altering the depth of the vectorizable loops (x direction) and the length of the OpenMP loops (z direction) we divided the workload between host and coprocessor only along the y direction. Individual host and coprocessor versions of all functions were written, and an offload in asynchronous mode was used in order to overlap work in host CPU and coprocessor.

When writing offload code for the Xeon Phi it is important to treat data transfers carefully. By default arrays that are offloaded get allocated on the coprocessor when the offload starts and de-allocated when the offload ends. This behavior is very different from the typical data management in CUDA codes, where by default data is persistent in the GPU between kernel calls. In the case of the LBM application we are working with we wish to avoid the overhead of allocating and de-allocating the offloaded arrays for every offload section. This is achieved by using optional arguments to the `offload` and `offload_transfer` directives that specify when to allocate and when to free arrays. We define the following macros for clarity:

```
#define ALLOC   alloc_if(.TRUE.)
#define REUSE   alloc_if(.FALSE.)
#define FREE    free_if(.TRUE.)
#define KEEP    free_if(.FALSE.)
```

For example:

```
!DIR$ OFFLOAD_TRANSFER TARGET(MIC:0) &
      IN( buff_mic : ALLOC KEEP )
```

Indicates that we wish to transfer the data inside array `buff_mic` to the coprocessor with ID 0, and that we wish to allocate this array, copy the data over, and then return to the host without freeing the array.

Similarly:

```
!DIR$ OFFLOAD BEGIN TARGET(MIC:0) &
      OUT(buff_mic : REUSE KEEP )
   CALL PostCollisionMIC
!DIR$ END OFFLOAD
```

Indicates that we wish to offload the function named `PostCollisionMIC`, and that we wish to transfer the contents of array `buff_mic` back to the host cpu without allocating the array on the mic again and without freeing it once the offload section is finished.

To illustrate the importance of not reallocating arrays we timed a series of offload transfers where data was sent to the coprocessor and then retrieved back from the host for arrays of different sizes. The results are shown in Figure 3. When the same array is reuse to perform the transfers we can see that for large messages the transfer rate reaches the usable limits of the PCIe bus, settling at slightly over 6 GB/s. In contrast, if each transfer has to be allocated and freed in the coprocessor the maximum effective bandwidth achieved is approximately 2 GB/s. This difference will clearly have a large influence in codes where offloads are performed multiple times inside a loop.

While writing this version of the code required a significant amount of coding time because of the need for separate CPU and MIC versions of most of the function calls in the code, it allowed for very fine tuning of the workload distribution between host and coprocessor. Since the native speed of the code on the MIC was more than twice as fast as that on the host we explored a range of workload fractions, assigning between 60% and 80% of the workload to the coprocessor.

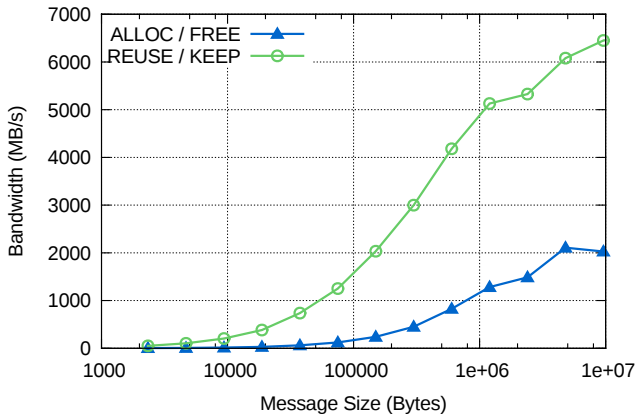

Fig. 3. Effect of reallocating arrays in data transfers to the Xeon Phi from the host CPU.

Using 16 threads on the host and 240 on the coprocessor it was found that optimal performance was achieved when 70% of the workload was offloaded to the Xeon Phi, achieving 142.8 MLUPS. This is equivalent to approximately 2.8 dual socket Xeon E5-2680 nodes.

While the performance of this version of the code was good, we had to introduce major changes in its structure as mentioned earlier. Creating duplicate versions of all functions for host and coprocessor is a simple procedure, but it is also a tedious and error prone procedure that extends the time spent on development. Also, this version of the code is limited to execution on a single node, so a more scalable solution would be desirable.

G. Symmetric execution

In order to explore a more scalable solution, a hybrid MPI/OMP version of the code was written, with partitioning allowed in the three spatial directions. This initial hybrid version of the code does not provide automatic workload balance between host and coprocessor, and relies on the static decomposition requested by the inputs to distribute the load. In this sense it is a less flexible code than the purely threaded version with offload, but it is also much simpler.

The ghost layer of cells that was previously used to correct streamed boundary values is used in this code to carry out the necessary data exchanges between MPI tasks. All MPI commands are executed outside threaded regions, or within `OMP MASTER` sections to avoid racing conditions.

While typically a single data exchange would be needed in a distributed single fluid LBM code, the multiphase model used in this work requires two data exchanges. These are performed in the `PostCollision` and `PostStream` functions. Partial superposition of communication and work is achieved by using asynchronous MPI calls (`MPI_Isend` / `MPI_Irecv` / `MPI_Waitall`) so that message exchange for each spatial direction overlaps with data packing for the others. This can, of course, be improved in multiple ways, but since we want to focus on the development for the Xeon Phi platform in this work we will analyze this simpler code and avoid further complication.

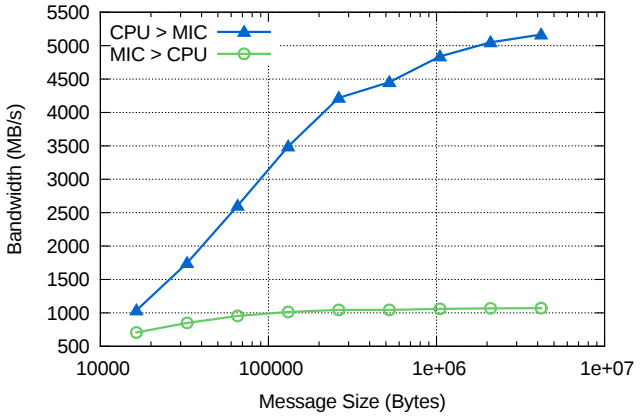

Fig. 4. Efective bandwith of MPI data exchanges initiated on the host and the coprocessor. Notice the large asymmetry in data transfer rates.

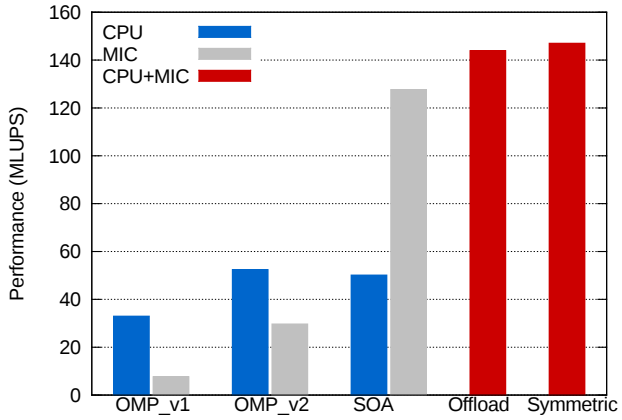

Fig. 5. Performance of all tested versions of the code on both host CPU and Xeon Phi coprocessor. Notice the excellent performance of the symmetric execution. The OMP_v1 case refers to using OMP in the `Collision` function only, while OMP_v2 corresponds to the parallelization of both the textttCollision and `Stream` functions. Of particular importance if the impact that of the change from AOS to SOA had on the vectorization efficiency and overall performance.

We were initially worried about the on-node performance of a symmetric execution of the LBM code because MPI microbenchmarks had shown a large difference in the MPI bandwidth for communications to and from the coprocessor. As shown in Figure 4 the MPI transfers from host to coprocessor were able to take good advantage of the PCIe bus bandwidth, but transfers from the coprocessor to the host seemed to achieve a limited 1 GB/s transfer rate. This microbenchmarks were run using a combination of `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` calls and run multiple times to confirm the different transfer data rates. While the difference in effective transfer rates was a concern, the time spent on data exchange for this code is minimal compared to the amount of work assigned to each task for any useful domain size and, as we will see shortly, it turned out not to be of critical importance. It is important to mention that these results were obtained using Intel MPI version 4.1.1.036, and that recent work from Potluri et al. [13] seems to significantly improve coprocessor to host data transfers in the Mvapich2 library [14], suggesting this will not be an issue in the near future.

From previous work we knew that the coprocessor outperformed the host by a factor approximately between 2:1 and 2.5:1, since native execution was 2.5 times faster on the coprocessor than the host and the offloaded version worked best when approximately 70% of the workload was given to the Xeon Phi. With this information in mind we chose a test case where we assigned 2 MPI tasks to the coprocessor and only one to the host. As shown in Figure 5 this provided even better performance than the purely threaded code, achieving a performance of 147.0 MLUPS.

Figure 5 also shows the tremendous improvement in performance from the original version of the code to the current symmetric version, with an improvement of over 19x in execution with respect to the original port of the code.

IV. Conclusions

We have described our efforts porting an LBM code to run on the Intel Xeon Phi coprocessor. Key performance improvements were obtained by changing from an AOS arrangement of the data to SOA, and by parallelizing functions with intense motion of data. It is important to highlight that timing information from execution profiles on the host was of little use in the porting process, since the bottlenecks changed when moving to the MIC architecture. Previously negligible sequential code sections became critical hotspots in the execution and required parallelization in order to achieve acceptable performance.

While a fully threaded version using asynchronous offloads achieved good performance, we determined that writing a hybrid code that worked in symmetric execution mode was more effective in terms of development time and future scalability.

The analysis of effects related to affinity indicates that using a `balanced` setting is optimal for memory intensive codes like the one tested. The LBM code ported achieved a performance equivalent to 2.5x dual E5-2680 socket nodes on native execution, and over 2.8x when run in symmetric mode using the host CPUs as well as the coprocessor. Overall this is a promising result when the relatively small changes introduced to the code are considered.

Future work involves the analysis and optimization of off-node performance, as well as strategies to reduce the number of simultaneous memory streams used in the code.

References

[1] http://top500.org/lists/2013/06/

[2] J. Jeffers and J. Reinders, Intel Xeon Phi Coprocessor High-Performance Programming, Morgan Kaufmann, Walthman, MA, USA, 2013

[3] G. Misra, N. Kurkure, A. Das, M. Valmiki, S. Das, and A. Gupta, Evaluation of Rodinia Codes on Intel Xeon Phi. 4th International Conference on Intelligent Systems Modelling & Simulation (2013) 415–419

[4] C.Q. Yang, W. Qiang, C. Cheng, K.Y. Wen, and Q. Jin, Accelerating PQMRCGSTAB Algorithm on Xeon Phi. Advanced Materials Research 709 (2013) 555–562

[5] E. Saule, K. Kaya, and U.V. Catalyurek, Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. arXiv preprint arXiv:1302.1078 (2013)

[6] K.W. Schulz, R. Ulerich, N. Malaya, P.T. Bauman, R. Stogner, and C. Simmons, Early Experiences Porting Scientic Applications to the Many Integrated Core (MIC) Platform. TACC-Intel Highly Parallel Computing Symposium, Tech. Rep., April 2012, Austin, TX, USA

[7] S.J. Pennycook, C.J. Hughes, M. Smelyanskiy, and S.A. Jarvis, Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. IEEE International Parallel & Distributed Processing Symposium, 20-24th May 2013, Boston, MA, USA

[8] http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide

[9] H.W. Zheng, C. Shu and Y.T. Chew, A lattice Boltzmann model for multiphase flows with large density ratio, J. Comput. Phys., 218 (2006) 353–371

[10] C. Rosales, D. Whyte and C. Ming, A massively parallel lattice Boltzmann method for large density ratios, 7th Asia CFD Conference Proceedings, 2007, Bangalore, India

[11] R.R. Nourgaliev, T.N. Dihn, T.G. Theofanous and D. Joseph, The lattice Boltzmann equation method: theoretical interpretation, numerics and implications, Int. J. Multiphase Flow 29 (2003) 117–169

[12] S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond, Oxford University Press, Oxford, UK, 2001

[13] S. Potluri, A. Venkatesh, D. Bureddy, K. Kandalla, and D.K. Panda, Efficient Intra-node Communication on Intel-MIC Clusters. IEEE International Symposium on Cluster Computing and the Grid (2013) 128–135

[14] MVAPICH2: High Performance MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/.